GPU ACCELERATED ADAPTIVE COMPRESSED SENSING

A Thesis
Presented to
The Academic Faculty

By

Idris O. Somoye

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science in Electrical Engineering

Georgia Institute of Technology
December 2016

GPU ACCELERATED ADAPTIVE COMPRESSED SENSING

Approved by:

Dr. Abhijit Chatterjee, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Arijit Raychowdhury
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Justin Romberg
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: December, 7th, 2016

ACKNOWLEDGEMENTS

First, I would like to thank God with whom nothing is truly impossible. Next, I would like to thank my advisor, Dr. Abhijit Chatterjee, whose support was invaluable throughout this process. I would also like to express my gratitude to Dr. Arijit Raychowdhury and Dr. Justin Romberg for serving on my reading committee. This work would also not have been possible without the mentorship and assistance I received from Josh Wells.

Lastly, I would like to thank my family. My mother Omolara, my brother Abdul-Azeez, and my fiancé Sariat have offered me nothing but unwavering support. For this I am deeply filled with gratitude and pride.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

SPC	Single Pixel Camera
GPU	Graphics Processing Unit
CS	Compressed Sensing
ACS	Adaptive Compressed Sensing
FPGA	Field Programmable Gate Array
CCI	CMOS Random Convolution Compressed Imager

SUMMARY

There are presently image sensors based around compressed sensing that apply the fundamental theory to video acquisition; however, these imagers require specialized hardware modules that are not widely available and therefore are not currently practical for video sensing. To deliver a practical image sensor that applies compressive sensing, I propose an imaging system based on a GPU and an off-the-shelf conventional image sensor that takes advantage of parallel computations for efficient transforming of data to the compressed sensing domain. This imaging system, by taking advantage of GPU processing along with straightforward communication methods between the host and the GPU, easily accommodates algorithms that rapidly change the sensing basis, making compressed sensing more applicable despite the general lack of hardware. Simulation results show that the GPU based compressive sensing imaging system delivers a viable and practical imager that is able to quickly compress images, providing a real-time video encoder for low power systems.

CHAPTER 1

INTRODUCTION

Not much work has been done on examining the advantages of computing on the graphics processing unit when it comes to encoding signals with compressed sensing. By capitalizing on advancements in GPU architectures and compressive sensing encoding techniques, this work aims to present an imager that provides a feasible encoding platform using conventional off-the-shelf sensors. This thesis demonstrates a practical implementation of parallel compressed sensing (CS) encoding with the use of graphics processing unit (GPU) programed with a specialized language called CUDA and a software package called Arrayfire.

1.1 Compressed Sensing

In recent years engineers have been pushing the boundaries of Shannon-Nyquist sampling theorem which states the number samples needed to reconstruct a given signal. This has led to a series of mathematical equations [7] and technical principles being incorporated into a field called compressed sensing [3]. Compressive sensing allows for a more efficient way to sample and reconstruct signals compared to traditional data acquisition and reconstruction techniques, which may under use samples.

Despite the extensive amount of applications, there isn't much literature on the implementation of CS encoding methods for the GPU. In this thesis I take advantage of the computational gain that the GPU provides to create and imager that performs the encoding of an image in a compressed sensing domain.

1.2 Graphics Processing Unit

Graphics processing units are highly parallel, multi-threaded, multiple core processors, which are gaining great popularity due in part to the large peak performances that the devices provide. Initially developed for graphics rendering, GPU's have come a long way and are now known as computational workhorses because of the peak computational performance and high memory bandwidth [18]. Furthermore, GPUs are now used in an array of applications such as a molecular dynamics [20], computational finance [24] and other applications [15]. Efficient implementations of numerical algorithms on the GPU are difficult to develop due to the complexity of the architectures and their technical specifications [22]. In order to overcome this hurdle, in this thesis the use of a software platform called Arrayfire created by AccelerEyes aids in generating GPU code [21].

1.3 Overview

The remainder of the thesis develops the key principles that will be needed to understand the work being presented. In Chapter 2, a more extensive background of compressive sensing is given along with some present encoding methods. Chapter 3 presents an overview of GPU's and the two GPU programming languages used in this thesis, along with several GPU code optimization techniques. I then provide an implementation which uses the theory of CS along with a GPU programed with Arrayfire and CUDA in Chapter 4. An introduction of NVIDIA's Jetson graphics board is presented in Chapter 5, as well as simulations and results. In the last chapter, I present my concluding remarks followed by possible directions in which this work may be taken.

CHAPTER 2

COMPRESSED SENSING BACKGROUND

Compressed sensing ((1], (2], (3]) is a signal processing technique for efficiently acquiring and reconstructing an undersampled signal by finding solutions to underdetermined linear systems. This is based on a principle that, through optimization, the sparsity of a signal can be exploited to recover said signal from far fewer samples than required by the Shannon-Nyquist sampling theorem.

This chapter concentrates on CS encoding structures, adaptive compressed sensing and present CS hardware imagers. CS signal encoding from a basic mathematical standpoint would assist in understanding the difference in the different signal acquisition methods, and a description of present CS hardware will help demonstrate the benefits of the GPU CS imaging system.

2.1 Compressive Sensing Encoding

At its core, CS can be viewed as a mathematical framework that studies accurate recovery of a signal represented by a vector of length $\mathbf{N}$ from $\mathbf{M} \ll \mathbf{N}$ measurements, ultimately performing compression and signal acquisition concurrently. In CS, the signal $\mathbf{x}$ is not acquired directly; instead $\mathbf{M} \ll \mathbf{N}$ linear measurements are acquired with Equation 1 below.

$$y = \Phi x \tag{1}$$

In this equation, we refer to $\mathbf{y}$ as the measurement vector, the samples obtained after compressed sensing.

Φ, in Equation 1, is a M × N CS sampling basis. When constructing a sampling basis, a core CS principle that must be considered when seeking an application is incoherence [3]. Incoherence states that unlike the signal being sensed, the sampling basis must have an exceedingly dense representation in the CS basis Ψ . As it turns out, random matrices are largely incoherent with any fixed basis Ψ, for this work a random Gaussian matrix is derived and stored in memory.

Sampling Basis (Φ) Signal(x) Sampling Operation Output (y)

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \end{Bmatrix} = \begin{Bmatrix} a_{11}b_1 + a_{12}b_2 + a_{13}b_3 \\ a_{21}b_1 + a_{22}b_2 + a_{23}b_3 \\ a_{31}b_1 + a_{32}b_2 + a_{33}b_3 \end{Bmatrix} = a_{ij}b_j$$

Figure 1: Compressed sensing sampling method

Figure 1 shows a representation of the sensing method as a Hadamard product, while a mathematical representation is shown in Equation 2.

$$y = \sum_{i}^{M} \sum_{j}^{N} x(j) * \Phi_M(i)(j) \quad (2)$$

From Equation 2, it is clear that as the number of random samples (M) increases the complexity and execution time of the problem also increases. Performing this operation on a single threaded processor would be very time consuming, however by using a multithreaded processor the sampling operation can be parallelized by running each M operation on individual threads, as shown in Figure 2.

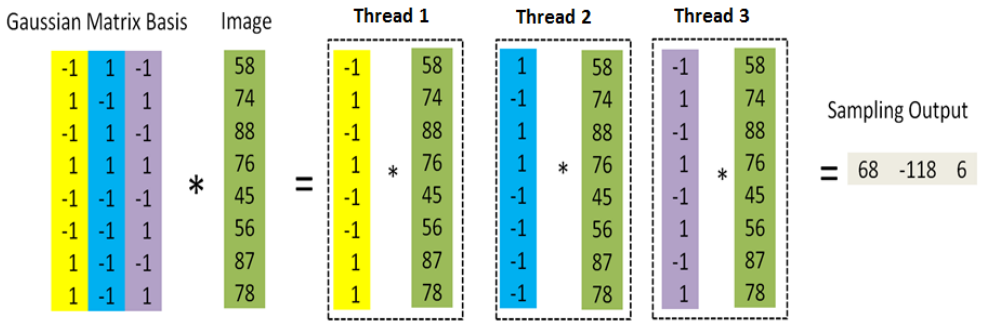

Figure 2 – Parallel image sampling example

This work's main interest in compressed sensing is in constructing an image sensor that exploits signal structure in order to reduce the sampling rate and subsequent demands on signal storage. In this approach, the relevant information contents of a signal are used to determine the sampling rate rather than the dimensions of the space in which the signal is captured from. With compressed sensing we transform an image from raw format in the spatial domain into a random basis domain where less data is required to represent the image (Figure 2).

2.2 Adaptive Compressed Sensing

ACS (Adaptive Compressed sensing) ((5], (6]) is a variation of CS that takes into account the time domain as well as the spatial domain. Normal implementations of compressive sensing cameras are able to compress the spatial dimensions but the fact that they need to make measurements in a sequential manner before the scene changes makes them inefficient for video imaging. To take full advantage of compressed sensing and the compressive sensing cameras, we must find ways to exploit the temporal sparsity that is usually common in images.

An application that implements ACS, called predictive video encoding (PVE) [26], is used in this thesis. PVE uses previously encoded information to form a model and predict how

upcoming information in the video should be encoded. The benefit being that predictive video encoding provides a real-time video encoding method, especially for low power mobile camera systems, without the high power consumption of modern video encoding standards such as H.264 and MPEG-4. The design reduces the power consumed by the entire video compression system by extending algorithm control all the way to the camera sensor itself. The objective of using predictive video encoding in this thesis is to implement an adaptive compressed sensing system with the GPU CS imager that reduces the amount of redundant information in a video signal, producing a more efficient and high-entropy signal.

2.3 Image Sensing Hardware for CS

Hardware designs have being developed for image sensors that exploit the compressive sensing theory. These designs are aided by the recent advances in CMOS technologies, while the feasibility of on-chip processing also continues to increase. As the GPU imager presented in this work is designed for encoding only, this section will cover only the encoding part of these CS imagers and will not discuss signal reconstruction. The advantages and disadvantages of these concepts will be discussed in addition to how they compare to our GPU imager.

2.3.1 Single Pixel Camera

Taking advantage of the CS theory is an image sensor called the “Single Pixel Camera” (SPC) [1]. The SPC has only one pixel to sense light, and records thousands of pixels one after the other to create an image. The SPC utilizes compressed sensing by first compressing the image data then recording it as opposed to conventional cameras that record millions of pixels then compress the data. This methodology, although seemingly counter-intuitive, drastically reduces redundant pixels. The SPC aims to address some of the difficulties of image acquisition,

the first being that when the desired image is not within the light wavelengths that silicon is sensible to, coincidentally same wavelengths at which humans see; image acquisition becomes much more expensive i.e. infrared sensors. Additionally the SPC resolves the data size problem in telecommunications as it reduces the amount of data representing a signal in situations where data transfer is expensive or slow. In both cases the SPC is a great solution as it minimizes cost per pixel and the compresses data needed to represent a scene.

The SPC however, has its shortcomings. First, it is extremely slow; taking minutes to construct a scene where conventional cameras can complete the same task in milliseconds and second, it is very rare and expensive to create.

2.3.2 CMOS Separable-Transform Image Sensor

In addition to the SPC, one of the other proposed CS image sensors is the CMOS separable-transform image sensor introduced in [12]. This image sensor is unique in its architecture as it compresses images in the analog domain before they are converted into the digital domain through a digital signal processor then remaps the image using analog processing. Rather than sensing the pixel values for the image, the transform imager projects the image on a specific basis and produces the projection coefficients.

The central ability of this imager is best described as a matrix transform $Y = A^T P_\sigma B$, where A and B are transformation matrices, Y is the output, P is the image, and the subscript σ denotes the selected sub-region of the image under transform. The image transformation and convolution are executed for distinct sub-regions of the image with 16 × 16 block sizes. Mathematically, the architecture computes the inner product of the incoming 2-D image with any matrix F that can be formed as the outer product of two vectors. The sensor performs the image transformation in two steps. The first step is focal plane processing along each column and

during sensing, a matrix multiplication using differential transistor in each pixel and Kirchhoff's current law along the columns. The results of the first matrix multiplication are fed an analog vector- matrix multiplier to perform the second step, which consists of matrix multiplications on the array, before the analog to digital converter. While this design offers great power savings and simultaneous sensing/compression, the hardware used is also extremely specialized and is not necessarily suited for widespread use.

2.3.3 CMOS Random Convolution Compressed Imager

In another method, called CMOS Compressed Imaging (CCI) [10], the sensing model applied is a random convolution strategy. In this design, M random samples are collected from the convolution of an image with a random filter. The fundamental part of this imager can be described as an array of N × N standard CMOS Passive Pixel Sensors (PPS). The image acquisition process is achieved in the steps described below.

In the first step a pseudo–random sequence is generated with a Linear Feedback Shift Register (LFSR) is generated up to N-bit and then stored in memory. Then the output current is measured for each PPS, the sign of this current is adjusted by the 1-bit value of the corresponding sequence digit residing in memory. The current is then collected (addition or subtraction) according to Kirchhoff's law on a wire connecting all pixels along each grid column. That output is converted to a digital value using an ADC which is in turn accumulated to form the final compressed image value. This process continues with slight variations until the system as the required number of samples for each frame. The system architecture of CCI has some drawbacks related to high power consumption, low sensitivity and increased difficulty in reconstruction as the sampling basis is not stored in memory. The system also assumes that the frame is completely still during sampling, making it impractical for most applications.

CHAPTER 3

GRAPHICS PROCESSING UNIT

Initially developed for fast and accurate rendering of graphics, GPUs could only be used through special graphics libraries. Recently, graphics processing units have moved away from simply rendering graphics. Today GPUs are widely used because of their large memory bandwidth and huge computational performance. A new field called general purpose graphics processing unit (GPGPU) has evolved to lead the application of GPUs for more varied purposes such as; molecular dynamics [16] and financial predictions [17]. The modern GPU is not only a powerful graphics engine but also a highly parallel programmable processor featuring peak arithmetic and memory bandwidth that substantially outperforms its CPU counterpart.

Although GPU's are now capable of delivering cost-effective and energy- efficient speed ups of common problems, the creation of efficient implementations remains a formidable task due to the complexity of the architecture and the technical specifications. To alleviate some of the hurdles in developing working and efficient GPU implementations, I propose the use of a software platform call Arrayfire, created by AccelerEyes, which will be addressed later in this chapter. Arrayfire is able to accelerate algorithms in C/C++ through the creation optimized kernels (GPU functions) written in CUDA.

3.1 CUDA

CUDA "Compute Unified Device Architecture" [(16], (17], (18]] is a parallel computing platform and programming model invented by NVIDIA which, released in 2007, is in fact two separate things. The first is a parallel computing architecture which deviates from the special

graphics libraries commonly used for interaction with the GPU. The second is a set of instructions used for the implementation of algorithms as an extension of the C language.

CUDA works as an intermediary between the CPU and the memory system associated with it, which is referred to as the 'host', and the GPU, with its memory will be referred to as the 'device'. To begin computation on the GPU the programmer must first allocate and set memory appropriately, due to the fact that the CPU and GPU have separate memory buffers. Memory is managed on the GPU through CUDA mainly through two function cudaMalloc and cudaMemcpy [16]. The first takes care of the memory allocation process while the second function transports the information from the host buffers to the device buffers.

Once data has been transferred to the GPU memory pool, the developer is able to use familiar tools to create functions called kernels that use parallel computation elements, known as threads. The procedure begins with the allocation of data both on the host and device, followed by copying of information from the host to the device. Then the GPU is instructed to perform computation and finally the result is copied back to the host from the device.

3.2 Advantages of CUDA

A great advantage of CUDA is the highly parallel nature of the architecture which has allowed the newer generation cards to redefine high performance computing [10]. This accounts for the capability of executing billions of calculations per second and is thus responsible for the surge of attention on GPU computing. Even with the possibility of tremendous computational gain, there are some drawbacks to using the GPUs. For example, the widely advertised speed ups and peak performance rates are usually not easily achieved. However, CUDA does allow the programmer a great deal of flexibility when implementing code, yet there exist some limiting factors. One such limiting factor deals with the performance rates where in order to reach the

largest Gigaflop counts, close to peak performance, computations should be carried out in single precision. The gap between the single and double precision computations may differ greatly depending on the GPU card being used. For my computations I use only single precision calculations.

3.3 Arrayfire

In general, programming on GPUs for scientific applications remains a difficult task due to the requirement that the user adjust to new programming paradigms. Arrayfire ((21], (23]) is a software platform developed at AccelerEyes that allows users and programmers to rapidly develop GPGPU applications in C, C++, Fortran, and Python.

Arrayfire is designed around a matrix holder object, the (array), which can hold many forms of data types including, floating point values (single and double precision), real or complex values and Boolean data. The Arrayfire (array) object also includes a wrapper that handles all the data transfers from CPU "Host" to GPU "Device". For comparison a standard cuda memory allocation is shown below:

```
int h_data[] = {1,2,3,4,5,6,7,8};
int *d_data = Null;
cudaMalloc((void**)&d_data, sizeof(h_data));
cudaMemcpy(h_data, d_data, sizeof(h_data), cudaMemcpyHostToDevice);
```

On the other hand, using Arrayfire the same code would look like:

```
int h_data[] = {1,2,3,4,5,6,7,8};
af::array data(8,1,h_data,afHost);
```

It's clear to see the advantage Arrayfire gives in usability and ease of development.

Arrayfire arrays are multidimensional, can be generated via simple matrix creation functions such as (ones, randu, etc.), and can be manipulated with arithmetic and functions. The great advantage of Arrayfire is that minimal knowledge and time requirement are needed for implementations making it very attractive to those with experience in CPU based programming that wish to use a Graphical Processor for scientific purposes.

3.4 Drawbacks to using Arrayfire

The key reason for using Arrayfire is to try and capture greater computational performance while not having to become an expert CUDA programmer. However, there are multiple factors which contribute to the speedups achieved with Arrayfire. The most obvious factor affecting the speedup is based on which one of NVIDIA graphics card is being used by the board. The more advanced the card, the greater the speedup one can achieve [16]. Also, my particular implementation is quite unique; if this were a simple matrix-vector multiplication on a single set of data, Arrayfire would probably perform very well. However, this problem requires constantly indexing into new data locations. Arrayfire's inability perform calculations could be a hindrance when compared to CUDA which lets the user implement this simultaneous indexing / computations.

3.5 Zero Copy

Arrayfire also has a major disadvantage in the specific GPU that we will be using. The Jetson TX1, which will be further described in Chapter 5, shares the same physical memory between the CPU and GPU. Therefore, with a tool provided in cuda called the "Zero Copy" we can access much faster host to device transfer times.

"Zero Copy" is a way to map host memory and access it directly over PCIe without doing an explicit memory transfer. It allows CUDA kernels to directly access the memory on the TX1 host side. Rather than reading data from global memory (limit ~200GB/s), data is instead read over PCIe and is therefore limited by PCIe bandwidth (up to 16GB/s).

For normal GPU applications where the device memory is separate from host memory zero copy offers no performance advantage. However, with the Jetson TX1 board equipped with a Tegra X1 processor this becomes very useful. The Jetson TX1 board has 4GB of physical memory that is shared by its ARM CPU and the NVIDIA CUDA GPU. If a cudaMemcpy Host->Device is done on the Jetson, the memory is simply being copied to a new location on the same physical memory and retrieving a CUDA pointer from it. For this reason zero copy is great in our application.

3.6 Dynamic Parallelism

To increase the performance of the GPU, dynamic parallelism is implemented in the CUDA code. Dynamic parallelism lets a kernel running on the GPU to invoke another kernel allowing for more adaptive parallelism and a more natural implementation of many parallel algorithms. Although there is an overhead cost of launching the children threads, for a problem like ours where we have for a large amount of blocks to process, dynamic parallelism is extremely beneficial.

CHAPTER 4

GPU CS IMAGER

As an alternative to the CMOS transform imager and the CMOS random convolution compressed imager (CCI), I propose a compressed sensing module embedded into or alongside the image sensor, directly downstream of the analog to digital converter (ADC). Rather than implementing random convolutions or analog computations, the GPU CS imager utilizes a Gaussian random matrix as a sensing basis and samples images in the digital domain using block-based compressed sensing [27]. This imaging system, using a conventional camera, allows for the real-time compression of larger images than the CCI is currently capable of while also requiring less complexity on the part of the sensor when compared to the CMOS Transform Imager.

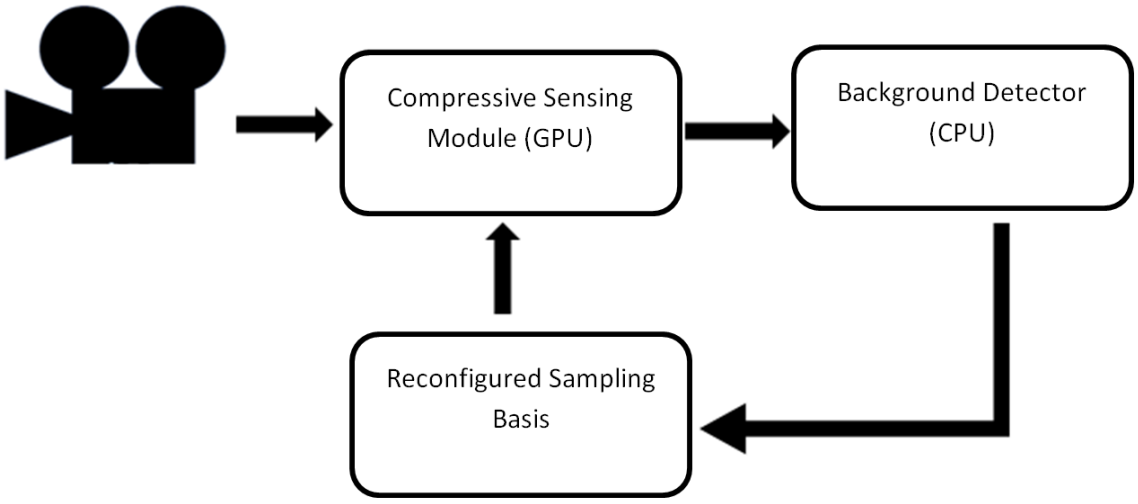

Figure 3: GPU CS imager

In my proposed solution matrix – vector multiplications are performed on the signal and random CS matrix using a dedicated hardware module, a graphics processor. The challenges in achieving this task both theoretically and in terms of hardware design can be reduced substantially when considering finite-dimensional problems in which the signal being measured can be represented as a discrete finite-length vector. In this arrangement the signal is a $N \times 1$ matrix while the sampling basis is a $M \times N$ matrix, with M being the number of samples being taken from the signal as represented in Figure 4.

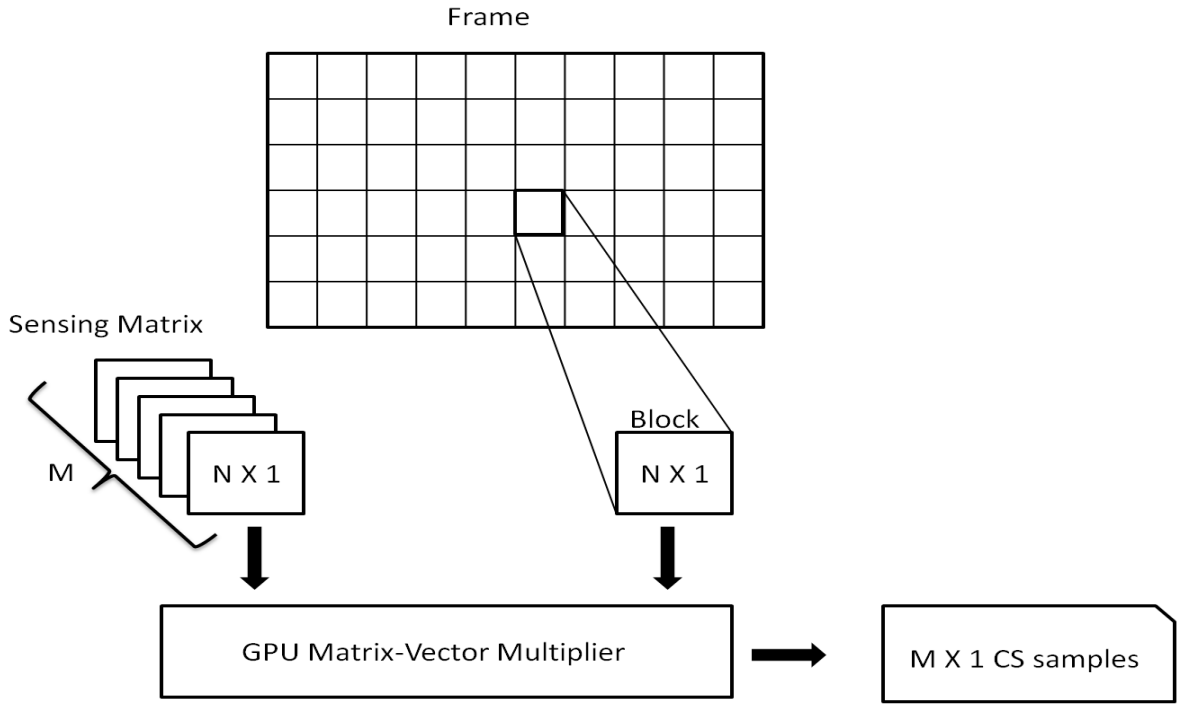

Figure 4: GPU Sampling Model

The sampling operation; usually performed on a digital signal processor or an analog processor, as described in Chapter 2, is of the form in Equation 2. In this case N represents the length of the block when unrolled and x is a N size array representing a sub-area of the frame.

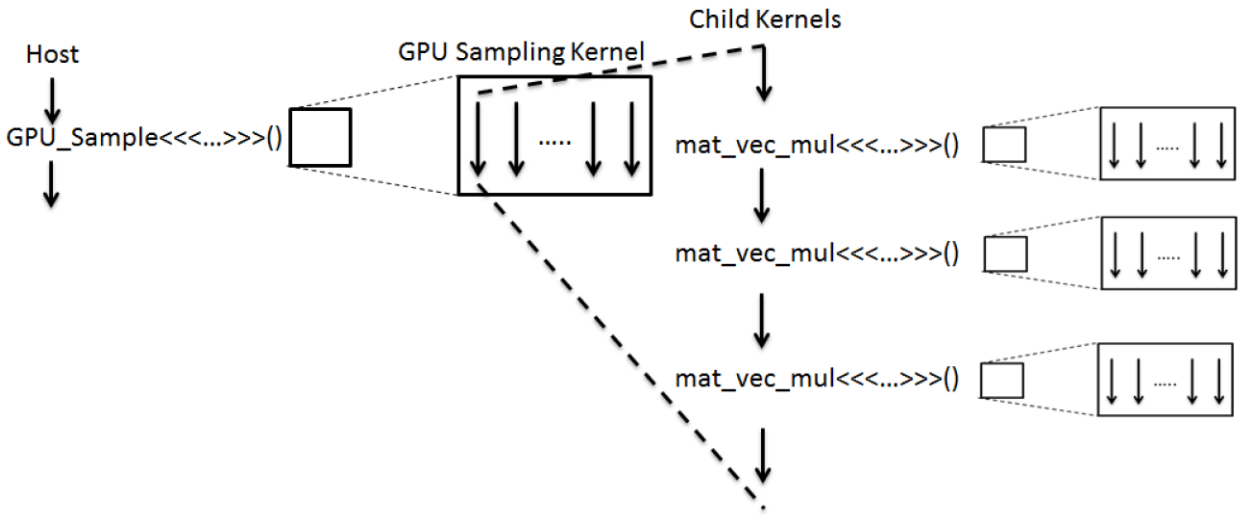

Figure 5: GPU dynamic parallelism

To perform the sampling operation in parallel each frame block is launched as an individual thread, then using dynamic parallelism, each thread launces many more child threads based on the blocks row size and the basis dimension. Each child thread loops through the basis dimensions, multiplying and accumulating across rows and columns. The result is an M length array representing the block in the compressed sensed domain.

4.1 Accelerated adaptive compressed sensing

An advantage of the GPU CS imager is its ability implement and accelerate compressed sensing applications such as adaptive compressed sensing. This work uses the GPU CS imager as a pre-compression camera sensor; applying a quad-tree based configurable basis and block sampling method [26] to improve the efficiency of compressed sensing with conventional cameras. In this system each incoming frame is handled as a collection of M X M blocks. Compressed samples are taken from each block and then background subtraction is performed on

the blocks in the compressed domain. The following frame is sampled based on the non-static block segments of the previous frame. In this method we take full advantage of the CS theory by exploiting spatial and temporal sparsity to capture fewer measurements than required in spaces that are decided to be irrelevant, and accumulating additional measurements from regions of an image that are essential. An example of the adaptive compressed sensing system performing background subtraction and object tracking is shown in Figure 6.

Figure 6: Background Subtraction in CS domain

4.1.1 Design Flow

To implement background subtraction based adaptive compressed sensing system on the GPU Imager, an application was developed that down-samples incoming frames on the GPU and returns compressively sensed samples to the host. The application receives a frame, a sampling basis, and a linked-list of block segment addresses and sizes. It then transfers the data as an array to the GPU using zero-copy. Raw image data is stored in pinned memory so no CudaMemcpy is

necessary (see code snippets in Appendix A and B for more information). The program performs down-sampling by iterating through the linked-list before transferring the resulting data back to the CPU. A major concern with the system is that the GPU could run out of memory in certain sequences due to the large image data being stored, for this reason dynamically allocated memory is carefully managed and calculations are performed to ensure that GPU memory is not overfilled when downsampling.

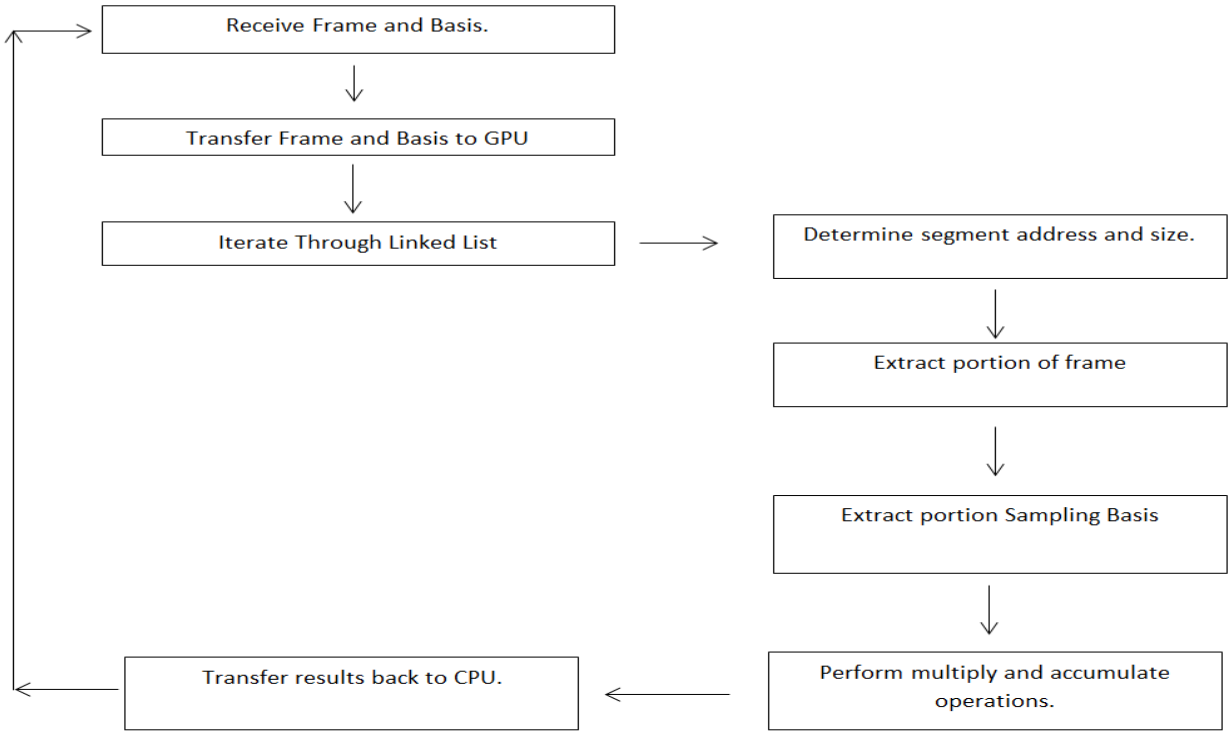

Figure 7: Design Flow Diagram

CHAPTER 5

RESULTS AND ANALYSIS

5.1 GPU Board

The GPU board used for the experiments is a Jetson TX1 [25]. It uses an NVIDIA GPU for graphics computations. The GPU used is a 256-core Maxwell GPU which provides 1TFLOPS of FP16 compute performance. The NVIDIA Jetson TX1 system-on-chip (SoC) (Figure 8) provides a graphics card within an embedded system equipped with an ARM processor and is a great fit for the problems addressed in this thesis.

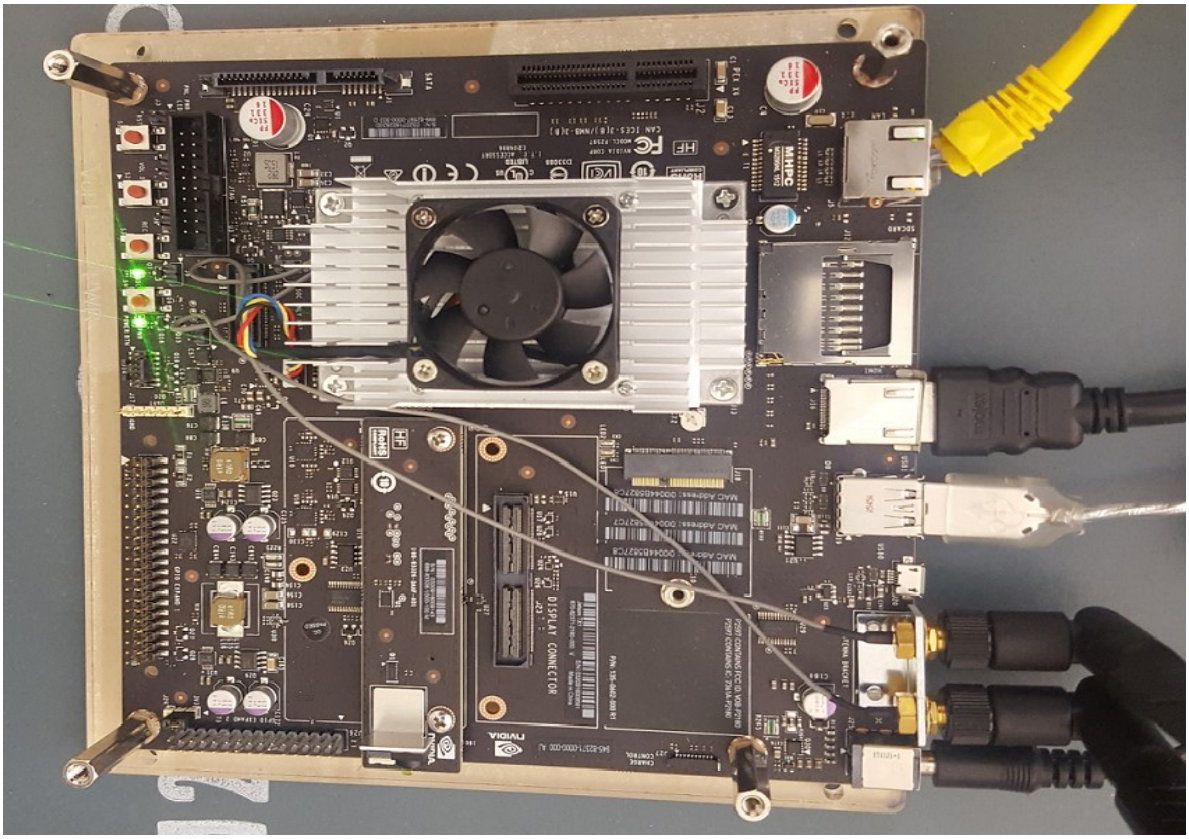

Figure 8 – Jetson TX1

The TX1 has 256 CUDA compute cores capable of operating at 998.4MHz. Along with its quad core ARM Cortex – A57 CPU it allows us to implement the system detailed in Figure 3 on a single board while typically consuming only between 8 to 10 watts, and up to 15 watts when fully utilized.

5.2 Parallel Computing

While C is an obvious choice for CPU programming languages in terms of performance, there are multiple languages to choose from when programing a GPU; such as CUDA, OpenCL and Arrayfire. While, all GPU programming languages have positives and drawbacks, the main objective is finding the one that works best with the particular application. With that in mind two different systems were created, one using Arrayfire and the other using native CUDA, the performances of both systems are discussed in the following sections.

5.3 Test Environment

The CPU used in executing all of the methods and thus being compared to the GPU is a Cortex A57 chip. These chips are quad core processors with a clock speed of 1.91GHz. These experiments were performed on NVIDIA's JETPACK v2.3.1 with the 64 bit Linux for Tegra v24.2.1 operating system platform. These results might be slightly different if performed on a 32 bit kernel build or a different compiler. The simulations were implemented using Arrayfire v3.4 and CUDA v7.0 respectively, with GCC v4.8.5 used for compiling.

5.4 Results

The results from the two GPU implementations of adaptive compressive sensing and the CPU only non-adaptive background subtraction and object tracking algorithms are presented in

this section along with a discussion of the simulations. The main points of comparison are between the run times for the C programming language implementations running on the host machine and the run times for the GPU compressed implementations created with CUDA and Arrayfire. Plots and tables showing run-time through frames of two video sequences, $(640\ X\ 360)$ and$(1920\ X\ 1080)$, are presented.

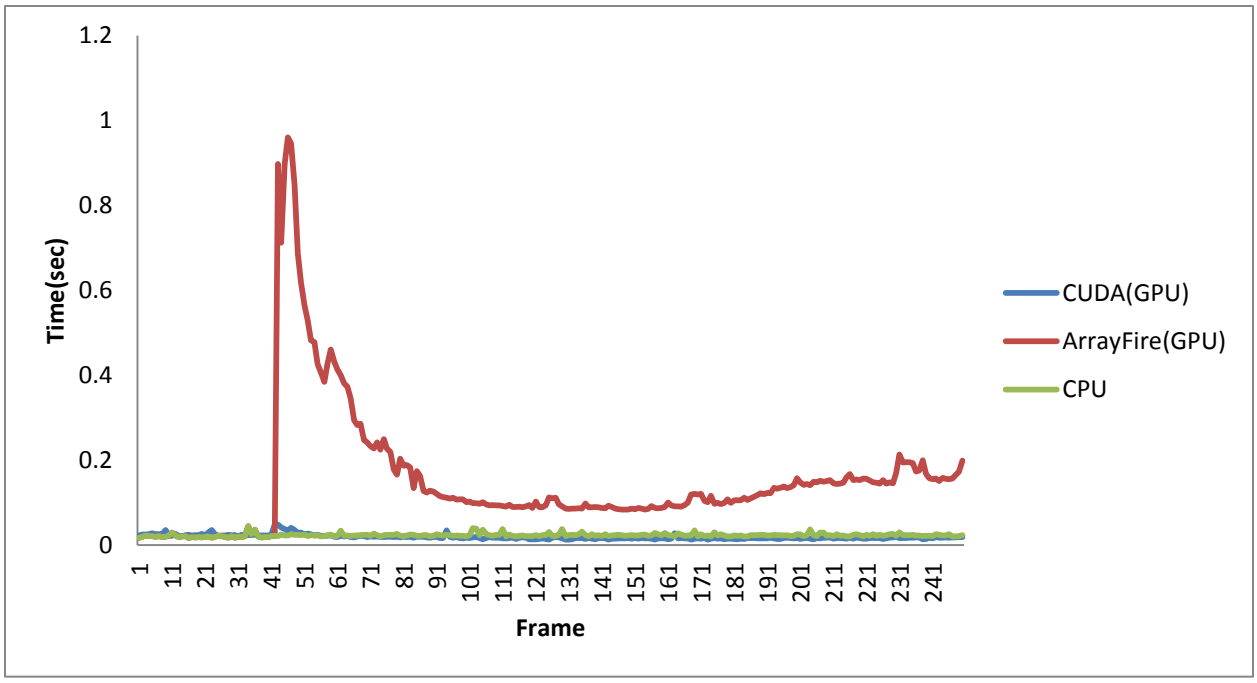

Figure 9: Background subtraction for a 640 X 360 video sequence

Figure 9 shows a comparison between a background subtraction on the GPU compressed image and a pixel level background subtraction. Arrayfire offers a disappointing return as it is much slower than the CUDA adaptive compressed sensing and CPU pixel level versions of background subtraction because of the large amount of time it spends compressing the image. This is due to the fact that Arrayfire code is not simultaneously performing computations on the frame segments as they are extracted leading to a high indexing overhead.

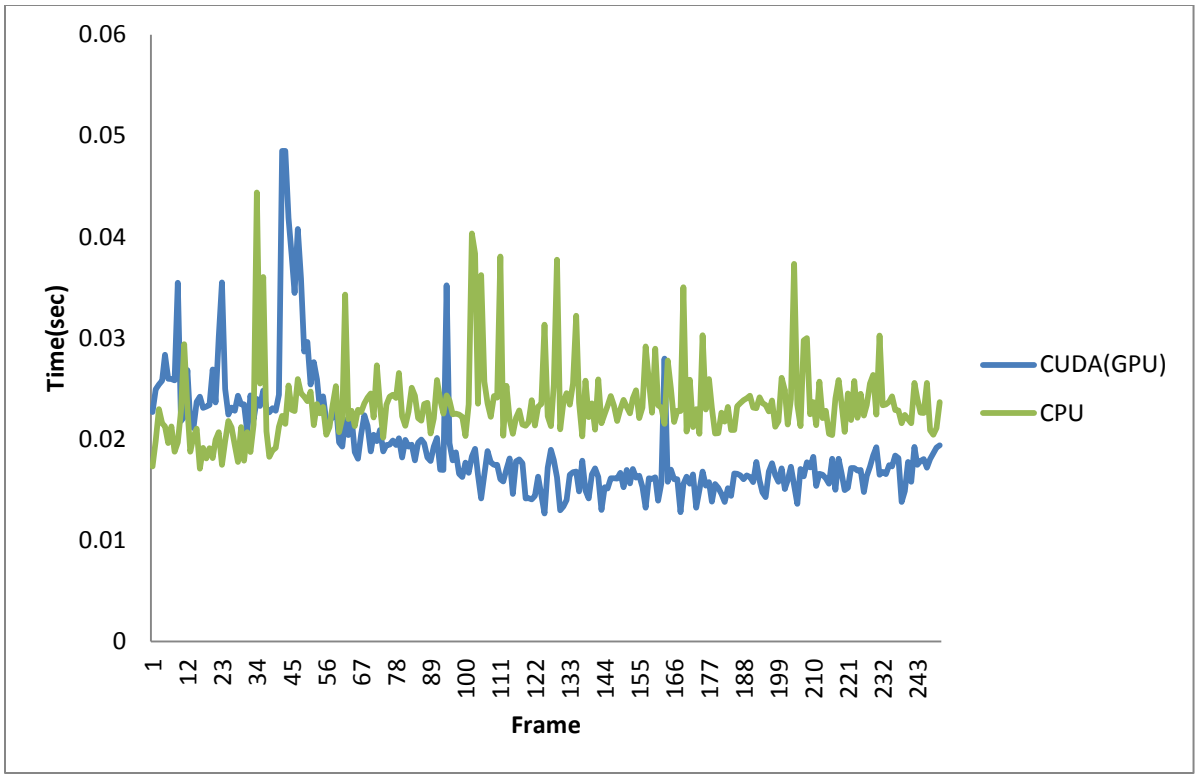

Figure 10: CUDA vs CPU 640 X 360 frame size

Figure 10 displays a clearer representation of the GPU performance by excluding Arrayfire's part. Over the course of the video sequence the CUDA code offers a noticeable increase over the CPU only version.

Table 1: Video sequence 1 GPU speedup

640 X 360 video speedup			
	CPU	CUDA(GPU)	Arrayfire(GPU)
Average time per frame (sec)	0.0235	0.018	0.152
Speedup over CPU	0	1.3	NULL

Table 1 displays the average time per second for each system. Pixel-level based background subtraction runs at an average of 43 frames per second, while adaptive compressed sensing based background subtraction runs at an average of 55 frames per second.

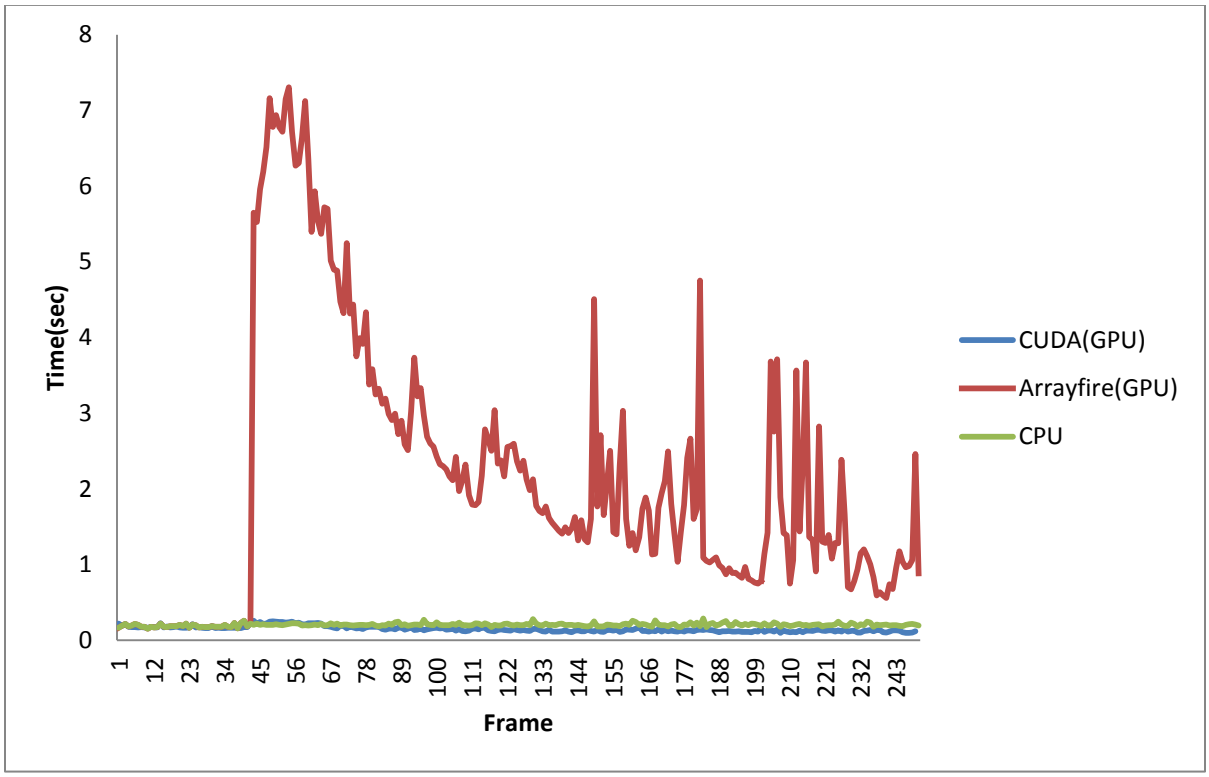

Figure 11 : Background subtraction for a 1920 X 1080 video sequence

Figure 11 shows the runtime for each background subtraction method on a larger frame size. As the frame size increases the Arrayfire version lags further behind. The overhead in indexing, extracting and then performing the inner product becomes too costly when the image is sufficiently large.

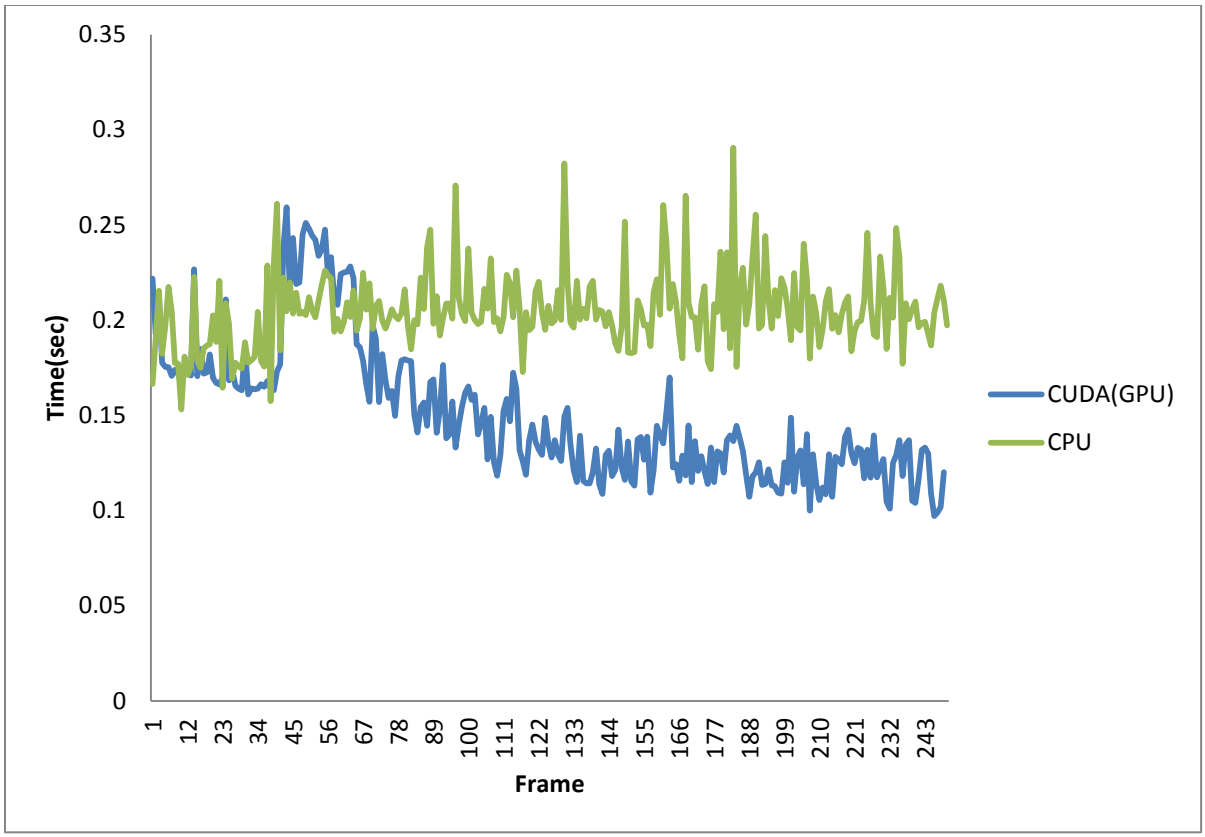

Figure 12: CUDA vs CPU 1920 X 1080 frame size

In Figure 12, Arrayfire's results are excluded from the plot leaving a better comparison of GPU vs CPU performance.

Table 2: Video Sequence 2 GPU speedup

1920 X 1080 video speedup			
	CPU	CUDA(GPU)	Arrayfire(GPU)
Average time sensing (sec)	.204	0.150	2.117
Speedup over CPU	0	1.3	NULL

Table 2 shows the average time per second on the larger frame size for each system. Pixel-level based background subtraction runs at an average of 4.9 frames per second, while adaptive compressed sensing based background subtraction runs at an average of 6.6 frames per second.

These results show that although Arrayfire allows for quick prototyping of scientific solutions on the GPU, CUDA is a better option when seeking acceleration of complex and unique problems. Also, while there is an increase in the CUDA based adaptive compressive sensing code the actual benefit of the system is in scenarios with multiple video streams. In such cases the multiple GPU CS imagers compress frames and then transfer the information to a central processor. Rather than analyze multiple full size images, the central processor only analyzes compressed data greatly reducing overall system compute time.

CHAPTER 6

CONCLUSION

This report explores the use of a graphics processing unit to improve the viability and applicability of compressed sensing imaging systems. Without a practical, accessible imager compressive sensing will continue to be a pertinent theory, yet unfeasible for widespread utilization. In order to address this problem a graphics processor has been employed to reduce the amount of overhead, in the form of computational time and power, needed to successfully implement compressive sensing techniques and algorithms on an off-the-shelf convention camera. The GPU CS imager allows applications to leverage compressive sensing in spite of the overall lack of hardware in the field. Also, due to the ease of communicating with the GPU, the GPU CS imager reasonably accommodates algorithms that rapidly change the sensing basis.

Exploiting the GPU CS imager, a solution for fast down-sampling of video frames using reconfigurable bases similar to the function of an embedded sensor has been proposed. Through simulations we can conclude that Adaptive compressed sensing using GPU CS imager provides a suitable application for compressed sensing as it increases the processing rate of several image processing systems with little added complexity in the form of the image sensor. One such image processing algorithm, background subtraction and object tracking, is addressed in this thesis to demonstrate the possible benefits the GPU CS imager.

6.1 Future Work

A possible direction for this work to continue would be to implement an FPGA into the design instead of a GPU. The benefit of using an FPGA is the ability to create an application specific hardware that contains only the necessary components to perform the desired operation allowing it to operate at a much faster rate. Compressed sensing in the digital domain using random convolution in hardware requires extensive "multiply and accumulate" (MACC) operations. Numerous MACC modules can be implemented on slices that are embedded into some moderns FPGA's. Another benefit of FPGAs is that the many SIMD units present in GPU's consume energy even if they are not in use, but if hardware can be configured to consist of only the necessary units then we can reduce power usage. The foremost issue with an FPGA implementation is developing a communication module between the FPGA performing the sampling operation and the CPU processing the compressed samples when the system requires a rapidly changing sensing basis.

APPENDIX A

ARRAYFIRE CODE WALKTHROUGH

```
void initGPU(void *objDetSys, unsigned char **host_frame, unsigned char **gpu_frame,int **host_basis , int **gpu_basis)
{
    ObjDetSys *sys = (ObjDetSys*) objDetSys;

    cudaSetDeviceFlags(cudaDeviceMapHost);
    cudaDeviceReset();

    /*Arrayfire Implementation*/
    double width = sys->frameSizePels[0]; //btemp->SampleBasis.dim1;
    double height = sys->frameSizePels[1];//;btemp->SampleBasis.dim2;
    double depth = sys->numCsDims;

    /*Allocate memory and store data on GPU device*/


    // Allocate host memory using CUDA allocation calls
    const size_t frameSize = sys->frameLenPels * sizeof(unsigned char);
    const size_t basisSize = sizeof(int)*width*height*depth;

    int *h_basis = (int*) malloc(
                sizeof(int) * width * height * depth);

    *host_frame = (unsigned char*) af::pinned(frameSize, s32);
    *host_basis = (int*) af::pinned(basisSize, u8);

    // Device arrays
    // Get device pointer from host memory. No allocation or memcpy
    cudaHostGetDevicePointer(gpu_frame,   *host_frame , 0);
    cudaHostGetDevicePointer(gpu_basis, *host_basis, 0);

    int m, n, i, c;
    c = 0;
    for (m = 0; m < depth; m++)
        for (n = 0; n < height; n++)
            for (i = 0; i < width; i++) {
                h_basis[c] = sys->csBasis[i][n][m];
                c++;
            }


    //copy new basis to gpu
    cudaMemcpy(*gpu_basis,h_basis,sizeof(int)*width*height*depth,cudaMemcpyHostToDevice);
    free(h_basis);
}
```

Code Snippet 1: Arrayfire GPU initialization

Code snippet 1 shows a sample of the code used to initialize and allocate pinned memory for employing zero-copy pointers with Arrayfire. Also, the snippet contains code to reset the GPU and properly configure the sampling basis.

```
CsBgModel *model = &node->model;

double xaxis = node->blkAddr[0] * sys->blockSize[0];
double yaxis = node->blkAddr[1] * sys->blockSize[1];

double xindex =  node->blkSize[0]*sys->blockSize[0];
double yindex =  node->blkSize[1]*sys->blockSize[1];
//assign temporary values for Downsampling

//perform downsampling in batch mode with arrayfire;
/********Operation of the DownSampling*************/
/** First the block in indexed into, then it is tiled along the 3rd dimension so it
 * matches with the sampling basis. Next, the matrix Multiplication is performed.
 * The result is then summed along the 3rd dimension and then the 2nd dimension.
 * The final result is a product with as many values as the number of samples required*/

//non JIT: Extract the section of the frame that is to be downsampled
/* tempFrame = tile(flat(transpose(F(seq(xindex) + xaxis,
                    seq(yindex) + yaxis))),1, depth).as(s32);*/



//non JIT: Extract the section of the Basis that would be used for downsampling
  /*tempBasis = moddims(transpose(S(seq(xindex) + xaxis,
                                  seq(yindex) + yaxis,seq(depth))),(xindex * yindex), depth).as(s32);*/

//non JIT: multiply and sum basis and frame, do not use
//displayed only to help in understading code.
//tempSum = flat(sum(tempBasis* tempFrame).as(s32))

//using JIT: combine frame and basis extraction into 1 operation, then sum.
 tempSum = flat(sum(tile(flat(F(seq(xindex) + xaxis,
            seq(yindex) + yaxis)),1, depth)* moddims(S(seq(xindex) + xaxis,
                    seq(yindex) + yaxis,seq(depth)),(xindex * yindex), depth)).as(s32));

//Transfer back to host
tempSum.host(&model->curSamples);
```

Code Snippet 2: Arrayfire GPU Downsample

The Arrayfire code takes advantage of Arrayfire's Just-in-time option as shown in code snippet 2. The sequencing of indexing into the frame, extracting each block, and performing the computations are left up to Arrayfire's at runtime to increase performance.

APPENDIX B

CUDA CODE WALKTHROUGH

```
void initGPU(void *objDetSys, unsigned char **host_frame,
        unsigned char **gpu_frame, int **host_basis, int **gpu_basis, int **host_data, int **gpu_data) {


    ObjDetSys *sys = (ObjDetSys*) objDetSys;

    cudaSetDeviceFlags (cudaDeviceMapHost);
    cudaDeviceReset();

    /*CUDA Implementation*/

    int basis_rows = sys->frameSizePels[0];
    int basis_columns = sys->frameSizePels[1];
    int basis_depth = sys->numCsDims;

    /*Allocate memory for basis transpose*/
    int *h_basis = (int*) malloc(
            sizeof(int) * basis_rows * basis_columns * basis_depth);

    // Allocate host memory using CUDA allocation calls
    cudaHostAlloc((void **) host_frame,
            sys->frameLenPels * sizeof(unsigned char), cudaHostAllocMapped);
    cudaHostAlloc((void **) host_basis,
            sizeof(int) * basis_rows * basis_columns * basis_depth,
            cudaHostAllocMapped);

    cudaHostAlloc((void **) host_data, BLOCK* MAX_CS_DIM * sizeof(int),
            cudaHostAllocMapped);

    // Device arrays
    // Get device pointer from host memory. No allocation or memcpy
    cudaHostGetDevicePointer(gpu_frame, *host_frame, 0);
    cudaHostGetDevicePointer(gpu_basis, *host_basis, 0);
    cudaHostGetDevicePointer(gpu_data, *host_data, 0);

    //transpose basis
    int m, n, i, c;
    c = 0;
    for (m = 0; m < basis_depth; m++)
        for (n = 0; n < basis_columns; n++)
            for (i = 0; i < basis_rows; i++) {
                h_basis[c] = sys->csBasis[i][n][m];
                c++;
            }


    memcpy(*host_basis, h_basis,
            sizeof(int) * basis_rows * basis_columns * basis_depth);

    free(h_basis);

}
```

Code Snippet 3: CUDA GPU Initialize

Code snippet 3 shows a sample of the code used to initialize and allocate pinned memory for employing zero-copy pointers with CUDA.

```
__global__ void mat_vec_mul(int rows,int cols, int index_rows, int index_cols,int srows, int scols,int *basis,
      unsigned char*frame, int *dst, int depth)
   {

      //assign thread id
      int tid = threadIdx.x + blockIdx.x * blockDim.x;;

      //loop through sampling basis 3rd dimension
      if (tid < depth) {
      int sum = 0;
      for (int i = 0; i < rows; i++) {
          for (int j = 0; j < cols; j++) {
              sum += frame[(i + index_rows) + (j + index_cols) * srows]
                      * basis[(i + index_rows) + (j + index_cols) * srows
                              + (srows * scols * tid)];
          }
      }
      dst[tid] = sum;
   }
}
```

Code Snippet 4: CUDA Vector-Matrix Multiplication

Code snippet 4 shows a sample of the code used to initialize and allocate pinned memory for employing zero-copy pointers with Arrayfire. Also, the snippet contains code to reset the GPU and properly configure the sampling basis.

```
__global__ void parallel_sample(int * addr_zero,int * addr_one, int * blksize_zero, int *blksize_one,
        int sys_zero, int sys_one, int *basis, int *Dst, unsigned char* frame, int depth,
        int count,int srows, int scols)
    {
        //assign thread id
        int tid = threadIdx.x + blockIdx.x * blockDim.x;

        //loop through nodes
        if(tid < count)
        {
            //set indexing values
            int rows = blksize_zero[tid] * sys_zero;
            int cols = blksize_one[tid] * sys_one;
            int index_rows = addr_zero[tid] * sys_zero;
            int index_cols = addr_one[tid] * sys_one;

            //call vector-matrix multiply as child funtion
            mat_vec_mul<<<depth/THREADS_SMALL+1,THREADS_SMALL>>>(rows,cols, index_rows, index_cols,
                srows,scols,basis,frame,Dst+(depth* tid),depth);

            //sychronize stream
            cudaDeviceSynchronize();
        }


    }
```

Code Snippet 5: CUDA dynamic parallelism parent kernel

As show in code snippet 5, the parent kernel launches multiple child kernels in the form of a vector-matrix multiplier to complete the task.

```
void cudaSample(void *objDetSys, unsigned char *frame, unsigned char *gpuframe,
        int *basis, int *hostbasis, int *host_data, int *gpu_data) {

    ObjDetSys *sys = (ObjDetSys*) objDetSys;
    Stats *stats = &sys->stats;
    clock_t startTime, stopTime;
    startTime = clock();

    const int depth = sys->numCsDims;
    const int src_rows = sys->frameSizePels[0];
    const int src_columns = sys->frameSizePels[1];

    //indexing pointers for parrallel sampling
    int *blksizeone = NULL;
    int *blksizezero = NULL;
    int *blkaddrone = NULL;
    int *blkaddrzero= NULL;

    int *d_blksizeone = NULL;
    int *d_blksizezero = NULL;
    int *d_blkaddrone = NULL;
    int *d_blkaddrzero= NULL;


    //allocate sufficient memory for linked-list-data to  array conversion
    cudaHostAlloc((void **) &blksizeone, BLOCK * sizeof(int),
            cudaHostAllocMapped);

    cudaHostAlloc((void **) &blksizezero, BLOCK * sizeof(int),
            cudaHostAllocMapped);

    cudaHostAlloc((void **) &blkaddrone, BLOCK * sizeof(int),
            cudaHostAllocMapped);

    cudaHostAlloc((void **) &blkaddrzero, BLOCK * sizeof(int),
            cudaHostAllocMapped);

    //get device pointers
    cudaHostGetDevicePointer(&d_blksizeone, (void *) blksizeone, 0);
    cudaHostGetDevicePointer(&d_blksizezero, (void *) blksizezero, 0);
    cudaHostGetDevicePointer(&d_blkaddrone, (void *) blkaddrone, 0);
    cudaHostGetDevicePointer(&d_blkaddrzero, (void *) blkaddrzero, 0);
```

Code Snippet 6: CUDA downsampling preparation

Code snippet 6 shows the how the system is prepared for liked-list to array conversion. Pinned memory is allocated so that zero-copy can be used to prevent memory copies.

```
//interger to keep count of nodes
int counter = 0;


/*Declare first node then iterate through remaining nodes*/
for (node = sys->activeNodes.lh_first; node != NULL;
        node = node->activeNodes.le_next) {

    if (node->pelModelActive) {
        //pixel level sensing. no parallelization required
        PelBgModel *model = node->pelModel;
        int n;
        for (n = 0; n < sys->blockSize[1]; n++) {
            int nF = n + node->blkAddr[1] * sys->blockSize[1];
            int nFOf = nF * sys->frameSizePels[0];
            int m;
            for (m = 0; m < sys->blockSize[0]; m++) {
                int mF = m + node->blkAddr[0] * sys->blockSize[0];
                model->curSamples[m][n] = frame[mF + nFOf];
            }
        }
    } else {

        //comrressed sensing, optimize for parallelization
        //convert linked-list to array
        blksizeone[counter] = node->blkSize[1];
        blksizezero[counter] = node->blkSize[0];
        blkaddrone[counter] = node->blkAddr[1];
        blkaddrzero[counter] = node->blkAddr[0];

        //set address of node samples to pinned memory
        node->model.curSamples = host_data+(counter*depth);

        //increment counter
        counter++;
    }
}

    //launch sampling kernel
parallel_sample<<<counter/THREADS_BIG+1,THREADS_BIG>>>(blkaddrzero,blkaddrone,
    blksizezero, blksizeone,sys->blockSize[0],sys->blockSize[1],basis, gpu_data,
    gpuframe, depth, counter,src_rows, src_columns);
```

Code Snippet 7: CUDA downsampling

The linked-list to array conversion as well as the dynamic parallelization parent kernel call, are shown in code snippet 7. The conversion retains only the data required for downsampling and ignores all other members of each node.

```
void freeGPUmem(unsigned char *host_frame, unsigned char *d_frame,
        int *host_basis, int *d_basis, int *host_data, int *gpu_data) {

    //free allocated memory and poiters
    cudaFreeHost(host_frame);
    cudaFreeHost(host_basis);
    cudaFreeHost(host_data);
    cudaFree(d_frame);
    cudaFree(d_basis);
    cudaFree(gpu_data);

    //reset GPU
    cudaDeviceReset();
}
```

Code Snippet 8: Free allocated Memory

Lastly, code snippet 8 shows the function that frees all memory used by the GPU, whether pinned host memory or a device pointer.

REFERENCES

[1] M. F. Duarte et al., "Single-Pixel Imaging via Compressive Sampling," in IEEE Signal Processing Magazine, vol. 25, no. 2, pp. 83-91, March 2008.

[2] Marco F. DuarteMark A. Davenport, Single-pixel camera. OpenStax CNX. Apr 15, 2011 http://cnx.org/contents/fd2e709a-dcd8-4336-9865-0cb2e944a8e6@4.

[3] E. J. Candes and M. B. Wakin, "An Introduction to Compressive Sampling," in IEEE Signal Processing Magazine, vol. 25, no. 2, pp. 21-30, March 2008.

[4] Dharmpal Takhar ; Jason N. Laska ; Michael B. Wakin ; Marco F. Duarte ; Dror Baron ; Shriram Sarvotham ; Kevin F. Kelly ; Richard G. Baraniuk; A new compressive imaging camera architecture using optical-domain compression. Proc. SPIE 6065, Computational Imaging IV, 606509 (February 02, 2006);

[5] Noor I, Jacobs EL; Adaptive compressive sensing algorithm for video acquisition using a single-pixel camera. J. Electron. Imaging.

[6] Cevher, Volkan. "Compressive Sensing for Background Subtraction." Digital Signal Processing at Rice. Springer Berlin Heidelberg, 18 Oct. 2008. Web. 05 May 2016. <http://dsp.rice.edu/publications/compressive-sensing-background-subtraction>.

[7] M. F. Duarte and Y. C. Eldar, "Structured Compressed Sensing: From Theory to Applications," in IEEE Transactions on Signal Processing, vol. 59, no. 9, pp. 4053-4085, Sept. 2011.

[8] N. Hurley and S. Rickard, "Comparing Measures of Sparsity," in IEEE Transactions on Information Theory, vol. 55, no. 10, pp. 4723-4741, Oct. 2009.

[9] S.E. Pinto, L.E. Mendoza and E.G. Florez, “Compressive sensing in FPGA and microcontroller”, in International Journal of Engineering and Technology, vol. 7, no. 6, pp. 2202-2206, Jan. 2016.

[10] L. Jacques, P. Vandergheynst, A. Bibet, V. Majidzadeh, A. Schmid, and Y. Leblebici, “CMOS compressed imaging by random convolution,” in IEEE Int. Conf. Acoustics, Speech, and Signal Proc. (ICASSP), Taipei, Taiwan, Apr. 2009, pp. 113–1116.

[11] A. Borghi, J. Darbon, S. Peyronnet, T. Chan, and S. Osher. A Compressive Sensing Algorithm for Many-Core Architectures. In G. Bebis, et al., editor, Advances in Visual Computing, volume 6454 of Lecture Notes in Computer Science, pages 678-686. Springer Berlin I Heidelberg, 2010.

[12] R. Robucci, J. D. Gray, L. K. Chiu, J. Romberg and P. Hasler, "Compressive Sensing on a CMOS Separable-Transform Image Sensor," in Proceedings of the IEEE, vol. 98, no. 6, pp. 1089-1101, June 2010.

[13] M. Dadkhah, M. J. Deen and S. Shirani, "CMOS Image Sensor With Area-Efficient Block-Based Compressive Sensing," in IEEE Sensors Journal, vol. 15, no. 7, pp. 3699-3710, July 2015.

[14] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips, "GPU Computing," in Proceedings of the IEEE, vol. 96, no. 5, pp. 879-899, May 2008.

[15] Enhua Wu and Youquan Liu, "Emerging technology about GPGPU," Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on, Macao, 2008, pp. 618-622.

[16] NVIDIA Corporation. What is GPU Computing'? http:/ jwww.nvidia.com/ object/GPU Computing.html.

[17] NVIDIA Corporation. NVIDIA Tesla C2050 / C2070 Datasheet, 2010. http:/ jwww.nvidia.com/object/tesla_productJiterature.html.

[18] NVIDIA Coporation. NVIDIA CUDA C Programming Guide. Version 3.2., October 2010.

[19] Wolfram Research Inc. CUDALink User Guide. http:/ /reference. wolfram.com/mathematica/ CUD ALink/ guide/CUDALink.html.

[20] N. Ganesan, M. Taufer, B. Bauer and S. Patel, "FENZI: GPU-Enabled Molecular Dynamics Simulations of Large Membrane Regions Based on the CHARMM Force Field and PME," Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, Shanghai, 2011, pp. 472-480.

[21] James Malcolm ; Pavan Yalamanchili ; Chris McClanahan ; Vishwanath Venugopalakrishnan ; Krunal Patel ; John Melonakos; ArrayFire: a GPU acceleration platform. Proc. SPIE 8403, Modeling and Simulation for Defense Systems and Applications VII, 84030A (May 3, 2012);

[22] A. Borghi, J. Darbon, S. Peyronnet, T. Chan, and S. Osher. A Compressive Sensing Algorithm for Many-Core Architectures. In G. Bebis, et al., editor, Advances in Visual Computing, volume 6454 of Lecture Notes in Computer Science, pages 678-686. Springer Berlin I Heidelberg, 2010. 10.1007 1978-3-642-17274-8_66.

[23] Yalamanchili, P., Arshad, U., Mohammed, Z., Garigipati, P., Entschev, P., Kloppenborg, B., Malcolm, James and Melonakos, J. (2015).ArrayFire - A high performance software library for parallel computing with an easy-to-use API. Atlanta: AccelerEyes.

[24] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. 2013. Accelerating financial applications on the GPU. In Proceedings of the 6th Workshop on

General Purpose Processor Using Graphics Processing Units (GPGPU-6), John Cavazos, Xiang Gong, and David Kaeli (Eds.). ACM, New York, NY, USA, 127-136.

[25] Franklin, Dustin. "NVIDIA® Jetson™ TX1 Supercomputer-on-Module Drives Next Wave of Autonomous Machines." Parallel Forall. NVIDIA Corporation, 12 May 2016. Web. 11 Nov. 2016.

[26] J. W. Wells, J. Natarajan, A. Chatterjee and I. Barlas, "Real-Time, Content Aware Camera -- Algorithm -- Hardware Co-Adaptation for Minimal Power Video Encoding," 2012 25th International Conference on VLSI Design, Hyderabad, 2012, pp. 245-250.

[27] James E. Fowler, Sungkwang Mun and Eric W. Tramel (2012), "Block-Based Compressed Sensing of Images and Video", Foundations and Trends® in Signal Processing: Vol. 4: No. 4, pp 297-416.